

Specifying Asynchronous Transfer of Control

Padmanabhan Krishnan^{1 2}

Peter D. Mosses³

Technical Report COSC 06/91

A reformatted version of the report is to appear as a paper in the *Proceedings of School & Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, Springer-Verlag, *Lecture Notes in Computer Science*. Citations should refer to the proceedings.

¹Department of Computer Science, University of Canterbury, Christchurch 1, New Zealand
email:paddy@cosc.canterbury.ac.nz

²Author supported in part by grant 1787123 from The University of Canterbury, Christchurch and in part by the Danish Research Council

³Department of Computer Science, Ny Munkegade Building 540, Aarhus University, DK 8000, Aarhus C Denmark email:pdm@daimi.aau.dk

1 Introduction

A principal requirement of a safety critical system is that it should be able to cope with errors and deficiencies in software and hardware. There are two main approaches in handling this viz., masking and recovery. Masking is usually achieved by replicating the hardware/software. One can either adopt strategies such as voting [Avi85] or treat part of the system as a shadow system and activate it when a fault occurs [HAH89]. Even if a subset of the components fail, the entire system can continue to function. The degree of replication depends on the criticality of the unit and the probability of failure. It is easy to see that such a technique cannot be adopted for large systems, as the cost would be prohibitively large. Recovery from hardware failures, usually results in reassigning the task on the failed unit to other unit(s) in the system. Recovery from software failures is achieved by transferring control to a recovery unit.

The general strategy for recovery can be described as follows. After a unit detects a malfunction, another unit is notified. The notified unit responds to the malfunction as soon as possible by taking appropriate action. The action it takes depends on the nature of the error and could affect other units in the system.

[Cri91] describes the various dimensions that are important in fault-tolerant computing. It does not appear to be possible to support all the issues directly in a single framework. However, one can provide a few primitives which can then be used to code the various detection/recovery techniques necessary. Asynchronous transfer of control is an important primitive and in this paper we concentrate on this aspect. As fault recovery is a high priority task, the communication between the detection unit and the handler is usually in the form of an interrupt. In this paper we describe a semantic framework for interrupts and show how different kinds of recovery actions can be specified. The model is an extension of the Action Notation [Mos90, Mos92], which supports various features including distributed computation (asynchronously communicating agents). However it does not support interrupts (or asynchronous transfer of control.)

This paper is organized as follows. In the next section we present a brief overview of the Action Notation. In section 3 our model for interrupts is described. In section 4 the change to the operational semantics of Action Notation is described. In section 5, we present a few examples using the extended notation.

2 The Action Notation

The aim of Action Semantics, which has evolved from Abstract Semantic Algebras [Mos82], is to allow descriptions of realistic programming languages. It uses the Action Notation to specify elementary actions and techniques for combining them. Actions are objects which when performed process information and are used to represent semantics of programs. Actions can be combined using the action combinators to derive a compositional semantics.

Actions are classified into the following facets: 1) Control 2) Functional 3) Declarative 4) Imperative and 5) Communicative. We give a brief and informal introduction to the above facets.

The control actions include **complete**, **diverge**, **fail**, **escape**, **commit**. **complete** is an action that always terminates, while **diverge** never terminates. The **fail** action indicates abortive termination and is used to abandon the current alternative. The **commit** action corresponds to cutting away all alternatives, while **escape** corresponds to raising an exception.

The combinators include **or**, **and**, **and then** and **trap**. **or** represents non-deterministic choice. An alternative to the chosen action is performed when the chosen action fails (unless a **commit** has been performed.) **and** is a combinator which performs two actions

with arbitrary interleaving. and then corresponds to sequential performance, while trap corresponds to handling the exception.

The functional actions process transient (as opposed to input/output) data and give/are given data. The actions include **give** D which yields the datum D, **regive** which gives any data given to it. **choose** D gives an element of the data of sort D. The principal combinator is **then**. A1 then A2 corresponds to functional composition, i.e., A2 is given the data produced by A1.

The declarative actions process scoped information. The actions include **bind** T to D, which produces a binding of token T to datum D and **rebind** which reproduces all the bindings it received. The combinators include **moreover**, **hence** and **before**. A1 moreover A2 corresponds to letting bindings produced by A2 override those produced by A1. A1 hence A2 restricts the bindings received by A2 to those produced by A1. A1 before A2 corresponds to letting bindings accumulate.

The imperative actions deal with storage (consisting of cells) which is stable information. The actions include **store** and **allocate**. The action **store** D1 in D2 stores the datum D1 in cell D2 while **allocate** D corresponds to the allocation of a cell of sort D.

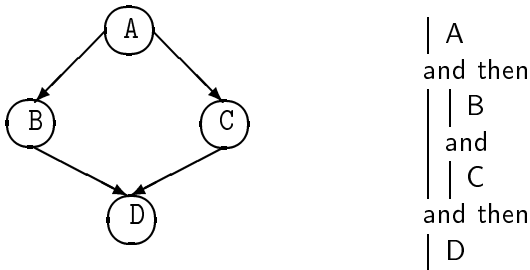
The action notation also provides primitives to model parallelism. Agents form the basic unit of parallelism. The actions for this facet include **send** D whose effect is to send the message identified by D, **receive** D whose effect is to receive any message identified by D and **subordinate** D which corresponds to creating a agent of sort D which is then sent a message containing actions which are to be executed. As agents cannot share cells, it models virtual nodes or distributed memory systems.

The Action Notation may appear informal, but it has a formal signature and an operational semantics specified in [Mos90, Mos92]. A brief introduction to the notation and its formal semantics is presented in [Mos89]. See also [MW87, Wat87].

3 Interrupts

Interrupts can be considered as a command to a scheduler directing it to execute a certain subprogram, viz., the interrupt handler. They can be classified as either hardware interrupts or software interrupts. A hardware interrupt can be thought of as a command to the 'instruction scheduler' and changes the program counter asynchronously. Therefore, the handling of a hardware interrupt suspends all processes on the device. A software interrupt on the other hand 'suspends' only the process for which the interrupt is intended. Conceptually, there appears to be little difference between hardware and software interrupts. However, if a distinction between distribution and interleaving is made a distinction between hardware and software interrupts is necessary.

In the Action Notation, an agent represents a processing element and actions executed by different agents can overlap in time. Hence, agents can be used to model hardware components of the system. The notation does not directly support the notion of processes (as in operating systems.) But unnamed processes can be modeled. For example, the fork-join structure (the figure on the left) can be represented as the action on the right.



Interprocess communication between these unnamed processes can occur via shared variables. It can also occur via message passing if each process receives messages only

of a particular sort and distinct processes operate on distinct sort of messages, i.e., the sort of message acts as process identifier. For example, B can execute `receive [For-B] message`, while C can execute `receive [For-C] message`. However, no direct naming scheme is supported by the notation.

Agents do not share memory and communicate solely via messages. That is, the Action Notation assumes a distributed memory model. Thus, hardware interrupts have to be modeled as messages. Modeling both hardware and software interrupts as messages gives a unified framework in which to study interrupts.

The interrupt handler can either be supplied by the unit raising the interrupt or can be fixed by the unit receiving the interrupt. As our aim is to describe fault-tolerant systems, we adopt the former option. The unit detecting the fault has a general idea of what went wrong and it pieces together a handler based on the information available. Thus, in our model interrupts are more like remote executions [SG90] than remote procedure call [BN81]. However, this is not a recommendation for an implementation strategy; rather, it should be considered to be a technique for specifying interrupts.

3.1 Hardware Interrupts

A hardware interrupt is modeled as a special sort of message. The message contains the interrupt name and the procedure to be executed as the handler (an abstraction). The receiving agent proceeds as normal till it receives an interrupt message. It then executes the handler contained in the message. The interrupt handler should have the power to terminate the current computation. One technique is for the handler to `escape` or to `fail` and to specify the continuation between the handler and the rest of the computation as `and then`. This, however, results in an abnormal termination for the entire computation. If the handler terminated normally and the continuation was `trap`, the entire computation terminates normally and the original computation is aborted.

An interrupt handler can be activated ‘asynchronously’ with respect to the rest of the computation. For example, consider the following action: `(A1 then A2)` and `(A3 then A4)`. While A1 and A3 are given the same transients, A2 and A4 receive their transients from A1 and A3 respectively. Consider the state where the action A1 has completed execution but not A3. In this state, the transients associated with A2 is not identical to the transients for A3 and A4. If an interrupt handler was invoked in this state and it has the power of altering the current transients, issues such as whether to discard the transients or overlay them need to be addressed. This unnecessarily complicates the semantics. To avoid these complications, the passing of transients and bindings from the handler to the rest of the computation is not supported. Therefore, if a handler is to affect the rest of the computation, it must alter the store or history (the stable state of the computation). If the continuation is `trap`, the original computation is resumed. Associated with the escape is a datum (identifying the cause), which is passed as a transient to the trap handler. As the original computation cannot receive any new transients, the data associated with the escape is lost. Thus escape in an interrupt handler is only a technique to restart the suspended process.

The semantics of such a computation is no different from the usual interleaving (`and`) semantics. From an implementation view point, this restriction is quite obvious. The transient data and binding represent register values and the execution of an interrupt handler requires the saving/restoring of registers and changes by the handler are to the memory.

As an interrupt handler indicates a high priority task, the handler must not be interleaved with the suspended computation. It must be finished before the rest of the task is resumed. Therefore, the continuation combinator for resumption is `and then`. The choice of the continuation combinator is made by the unit generating the interrupt as part of the

interrupt message.

Note that if interleaving is to be permitted one can use the **and** combinator. However, the **and** combinator is not ‘fair’ due to which the intuition behind interrupts is lost. Such behavior can be simulated in the current version of the notation by sending a message which is removed and enacted by a polling loop. But the execution of this message (handler) is not guaranteed.

Usually, masking and unmasking accompany interrupts. Masking of a particular interrupt allows executing a piece of code without being affected by that interrupt, while unmasking makes the code interruptible. Masking is necessary for 1) atomicity and 2) predictability. Certain code fragments (such as data-base updates) may represent critical sections and should be executed ‘atomically’. This can be achieved by masking all interrupts before executing the code and resetting them on completing the critical section. In real-time systems predictability is an important concern. Therefore, it is essential that an interrupt is not handled during a time critical computation.

As masking/unmasking is not supported in the Action Notation we define the following extensions. Define **mask D** (where D is a set of interrupt names) as setting a mask for all interrupts in D and **unmask D** as resetting the mask for interrupts D.

The operational semantics of interrupts should require that as soon as an unmasked interrupt message is detected, the agent’s normal processing is suspended and the interrupt handler activated. To identify a message as a hardware interrupt we define a sort restriction **interrupt _**. For example, **[interrupt][From-Disk] message** identifies a sort of hardware interrupts called **From-Disk**. The operational semantics of the Action notation does not have a construct which forces the presence of an item in the buffer to execute an action. Hence, one has to change the operational semantics to force the execution of the handler. To avoid race conditions, the activation of an interrupt handler masks interrupts of the same name. This prevents the handler(s) from getting interrupted by the same interrupt before it can do any useful work. If the handler is to be interrupted by the same type of interrupt it explicitly unmask the interrupt.

3.2 Software Interrupts

While hardware interrupt messages identify the agent to be interrupted, software interrupts only identify the sort of messages. There is no analog of names for processes. To identify processes which can be interrupted by software, we extend the action notation to include **listening to D A**, where D is a set of interrupt names and A the action. Intuitively, the execution of action A can be interrupted by any software interrupt in D. That is, the execution of **listening to D A** makes the execution of the action A sensitive to the interrupts named in D. As in the hardware case, the continuation can be **and then** or **trap**. Note that nesting of **listening to** is not the same as **listening to** of the union of the signal names. For example,

$$\begin{array}{l} \text{listening to D1} \\ \text{listening to D2 A} \end{array} \neq \text{listening to (D1} \cup \text{D2) A} .$$

This is because if a signal in D2 occurs the handler in the nested case can be preempted by a handler for a signal in D1. This is not the case in the union case.

A software interrupt should interrupt only the process for which it is destined. Therefore, the arrival of a software interrupt at an agent does not immediately force an asynchronous transfer of control to the handler. Only when the relevant process is executed is it interrupted. To identify a message as a software interrupt define a sort restriction **signal _**. For example, **[signal][kill-9]message** defines a sort of software interrupts called **kill-9**.

Software interrupts do not have the notion of masking and unmasking. Processes are susceptible to signals only if they indicate so. To avoid race conditions in software

interrupts, the handler is impervious to signals unless it explicitly exposes itself. However, the original process does not lose its ability to be interrupted on resumption. This is because unlike hardware masking, software ‘masking’ does not change the stable state.

In the next section, we describe the changes to the operational semantics to support interrupts.

4 Operational Semantics

The main features of the operational semantics for the Action Notation are as follows. The global state of the distributed computation is captured by an entity of the form **processing** $C\ S\ E$, where C is the state of the communication medium (i.e., the messages that are sent but not yet delivered), S is called the **stating** component and represents the state transition(s) being performed and E is the set of agents that are active. The local state of an agent is denoted by **state** $A\ s\ h$ where A is the action being executed along with the transients and bindings, s the storage and h the history. **step** $A\ s\ h\ c$ which changes the state to $A\ s\ h$, and sends the message in c represents a local transition. **stepped** given a state gives the next **step**. An auxiliary function **propagated** is defined which handles the propagation of transients and bindings and termination details.

For example, the following rules help to define the semantics for **and**.

- (1) **stepped** state $A1\ s\ h$:- **step** $A1'\ s'\ h'\ c'$ \Rightarrow
stepped state $\llbracket A1\ \text{“and”}\ A2 \rrbracket$:- **propagated** **step** $\llbracket A1'\ \text{“and”}\ A2 \rrbracket\ s'\ h'\ c'$
- (2) **stepped** state $A2\ s\ h$:- **step** $A2'\ s'\ h'\ c'$ \Rightarrow
stepped state $\llbracket A1\ \text{“and”}\ A2 \rrbracket$:- **propagated** **step** $\llbracket A1\ \text{“and”}\ A2' \rrbracket\ s'\ h'\ c'$

Recall that the **and** combinator defines the interleaved execution of two actions. The first rule states that if the state $A1\ s\ h$ can make a transition to the state $A1'\ s'\ h'\ c'$, $\llbracket A1\ \text{“and”}\ A2 \rrbracket\ s\ h$ can make a transition to $\llbracket A1'\ \text{“and”}\ A2 \rrbracket\ s'\ h'\ c'$. The second rule specifies the progress of $A2$. Note that the ‘:-’ can be interpreted as ‘ \rightarrow ’ as in labeled transition systems.

4.1 Hardware Interrupts

The main transition rule for an agent is as follows.

- (1) **stepped** state $A\ s\ h$:- **step** $A'\ s'\ h'\ c'$;
communications of c' :- C' ;
stating state $A'\ s'\ h'$:- S'
 \Rightarrow **step-stating** **processing** C **set** (state $A\ s\ h$) E :-
processing **union**(C, C') **set**(S') E .

The idea is that given a local state consisting of acting A with state s and history h , which can make a transition to state $A'\ s'\ h'$ and communicate C' , the global state is changed appropriately. To support interrupts the above transition rule is divided into two rules. The first transition rule is as above but with the additional check that the history has no pending interrupt message that can be handled, while the second rule activates an interrupt handler that is present in the buffer.

Before describing the transition rules, we need to address the issue related to masking. The masking vector is modeled as a cell (**masking-vector**), which can contain a set of interrupt names. The masking vector cannot be modeled as a transient or a binding as they are scoped information. The masking vector should be visible in all scopes and hence a part of the stable state. Masking an interrupt has the effect of adding the interrupt to the

stored values, while unmasking removes the interrupt from the stored set. The transition rules for masking/unmasking are

- (1) evaluated $D \ t \ b \ s \ h \text{ :- } si \text{ : set};$
 $m' = \text{union} (s \text{ at masking-vector}) \ si;$
 $s' = \text{overlay}(\text{map masking-vector } m', s)$
 $\Rightarrow \text{stepped state } \llbracket \llbracket \text{mask } D \rrbracket \ t \ b \rrbracket \ s \ h \text{ :-}$
 $\text{step } \llbracket \text{completed empty-map empty-map} \rrbracket \ s' \ h \text{ committed}$
- (2) evaluated $D \ t \ b \ s \ h \text{ :- } si \text{ : set};$
 $m' = \text{difference} (s \text{ at masking-vector}) \ si;$
 $s' = \text{overlay}(\text{map masking-vector } m', s)$
 $\Rightarrow \text{stepped state } \llbracket \llbracket \text{unmask } D \rrbracket \ t \ b \rrbracket \ s \ h \text{ :-}$
 $\text{step } \llbracket \text{completed empty-map empty-map} \rrbracket \ s' \ h \text{ committed}$

The main transition rule for an agent supporting interrupts is as follows.

- (1) $m = (s \text{ at masking-vector}) ;$
 $\text{nothing} = [\text{not in } m] [\text{interrupt}] \text{ message } \ \& \ \text{buffer of } h;$
 $\text{stepped state } A \ s \ h \text{ :- step } A' \ s' \ h' \ c;$
 $\text{communications of } c' \text{ :- } C' ;$
 $\text{stating state } A' \ s' \ h' \text{ :- } S'$
 $\Rightarrow \text{step-stating processing } C \text{ set } (\text{state } A \ s \ h) \ E \text{ :-}$
 $\text{processing union}(C, C') \text{ set}(S') \ E$
- (2) $m = (s \text{ at masking-vector});$
 $X: [\text{not in } m] [\text{interrupt}] \text{ message } \ \& \ \text{buffer of } h ;$
 $H = \text{Body} (\text{contents of } X);$
 $\text{Cont} = \text{Continuation} (\text{contents of } X);$
 $I = \text{Name} (\text{contents of } X);$
 $h' = \text{remove} (X, h);$
 $s' = \text{overlay}(\text{map masking-vector to union}(m, I), s);$
 $\text{stating state } \llbracket \llbracket \llbracket \text{enact } H \rrbracket \text{ empty-map empty-map} \rrbracket \text{ Cont } A \rrbracket \ s' \ h' \text{ :- } S'$
 $\Rightarrow \text{step-stating processing } C \text{ set } (\text{state } A \ s \ h) \ E \text{ :-}$
 $\text{processing } C \text{ set}(S') \ E$

4.2 Software Interrupts

The transition rules local to a ‘process’ are of the form $\text{stepped state } A \ s \ h \text{ :- step } A' \ s' \ h' \ c'$. As a software interrupt does not affect an ‘unarmed’ process, these rules need not be changed. We have to add rules to handle listening to $D \ A$ which before executing A checks the current buffer for the presence of a relevant software interrupt.

- (1) $\text{stepped state } A1 \ s \ h \text{ :- step } A' \ s' \ h' \ c';$
 $\text{nothing} = [\text{in } D] [\text{signal}] \text{ message } \ \& \ \text{buffer of } h;$
 $\Rightarrow \text{stepped state } \llbracket \text{listening } D \ A1 \rrbracket \ s \ h \text{ :- step } \llbracket \text{listening } D \ A' \rrbracket \ s' \ h' \ c'$
- (2) $X: [\text{in } D] [\text{signal}] \text{ message } \ \& \ \text{buffer of } h;$
 $H = \text{Body} (\text{contents of } X);$
 $\text{Cont} = \text{Continuation} (\text{contents of } X);$
 $h' = \text{remove} (X, h);$
 $\Rightarrow \text{stepped state } \llbracket \text{listening } D \ A1 \rrbracket \ s \ h \text{ :-}$
 $\text{step } \llbracket \llbracket \llbracket H \rrbracket \text{ empty-map empty-map} \rrbracket \text{ Cont } \llbracket \text{listening } D \ A1 \rrbracket \ s \ h'$

(3) stepped state \llbracket listening D \llbracket completed t b \rrbracket s h :- propagated step \llbracket completed t b \rrbracket s h

If there is no software interrupt, the ‘process’ continues to execute as usual. The presence of a relevant software interrupt activates the handler. The handler is given no datum or bindings (the **empty-maps**) to make the execution ‘predictable’. The last rule specifies the termination behavior of the process.

In the next section we show how the extended Action Notation can be used. As the Action Notation has been primarily designed to define semantics of programming languages, we concentrate on language constructs which can be used in fault-tolerant systems.

5 Examples

Two examples are considered here. The first is the modeling of heart beats [KU87]; a simple technique in fault detection and recovery. The second is a semantics for the asynchronous ‘and’ suggested as an extension for Ada [RTA88].

Both these examples use time outs. This requires the specification of time in the notation. While the notation uses a definition of time for its operational semantics, it does not give access to the current time at the notation level. This is to obtain algebraic laws such as **complete and then A** is equal to **A**. If the access to time were allowed, **complete and then give current-time** will not be equal to **give the current-time**. This can be rectified by defining that the action **complete** takes 0 time but then one can do infinite actions in 0 time. Even if time were available, one cannot specify a timeout for an action **A** as **A and time-out** as the **and** is not fair. Therefore, the **time-out** action may never be executed. Thus we define our own definition of time and code time outs as necessary.

We model time as an agent, which broadcasts the ‘current time’ to the relevant agents. For this to map to the usual notion of time, the execution of broadcasting and the message transfer time must be ‘regular’. The behavior of a clock agent starting from an initial value of time and a fixed increment of time can be specified as follows. The clock agent first receives a message containing a list of agents which require a time service after which a message of sort **Time** containing the time is sent periodically.

```
Metronome Init Incr =
| receive a message then
|   | bind %system-agents to contents of it
|   moreover
|   | bind %current-time to Init
| hence
|   unfolding
|   | Broadcast-time and then Step-Time
|   hence
|   | unfold .
```

```
Broadcast-time =
give the datum bound to %system-agents then
| unfolding
| | check (it is the empty-list)
| or
| | check (it is not the empty-list) and then
| | | give the datum bound to %current-time then
| | | | send [to (head it)][Time][containing the datum] message and then
| | | | give (the tail of the list) then unfold .
```


Step-Time = give the datum bound to %current-time then
 | give the sum (Incr, the datum) then
 | | rebind moreover bind %current-time to it .

A local agent can obtain the current time by selecting the maximum of all the values of Time messages in the buffer and is specified below. We do not insist that the messages are deleted from the current buffer as various messages (from potentially different time agents) could be used to create a distributed time reference and specify clock synchronization [CAS86, ST87].

L is the empty-list \Rightarrow received-time L = 0 .

L is list(m:[Time]message) ; T: natural is (contents of m) \Rightarrow received-time L = T .

L is concatenation(l₁,l₂) \Rightarrow received-time L = maximum(received-time l₁, received-time l₂) .

current-time = received-time [Time] current-buffer .

5.1 Heart Beats

Heart beats [KU87] or watch dogs [KK88] is a common technique for fault detection. In this example we show how this technique can be modeled. We assume that there is a main process which needs service from another process which is replicated on a number of service agents (such as Proc1 and Proc2 etc). We assume that the computation starts by using Proc1. Furthermore, the standby agent to be used when the agent currently in use fails is determined from the current agent by a function Next. There is also a heart beat agent or a watch dog process, (HBC) which periodically sends a message to the service agent currently in use and delays for time Timeout. If an acknowledgement from the service agent is received, the heart beat agent continues as usual. However, if no acknowledgement is received, it assumes the service agent is no longer usable and thus interrupts the main process to reconfigure to use the standby process.

We define MPB as the main process, which initializes the system by storing the agent name Proc1 in the cell %service-agent and then executes the code MPC. MPC, in our example is an infinite loop consisting of performing an initial computation (indicated by Local-Task-1) followed by getting service and using the result obtained (indicated by Local-Task-2.) As we concentrate on the fault-tolerance aspect of the system and not on the computational aspects, we do not specify a behavior for Local-Task-1 and Local-Task-2. This is indicated by defining their behavior to be \square . We specify a system where obtaining a service is atomic with respect to reconfiguration. (More elaborate schemes can be defined by generalizing the state information and the recovery mechanism.) Assume that MPB is executed on an agent called MP. The heart beat code (HBC) sends a Poll message to the service agent and then awaits a reply within time Timeout. If the timer expires, the waiting for acknowledgement is terminated (by the escape) and the MP agent is interrupted with the Reconfigure message. The message is an abstraction which when enacted alters the name of the service agent. The ‘continuation’ is and then as the original computation need not be abandoned.

Initialize = store Proc1 in %service-agent

MPB = Initialize before MPC

MPC = unfolding
 | Local-Task-1 and then
 | | Get-Service then
 | | | Local-Task-2 and then unfold

Local-Task-1 = □

Local-Task-2 = □

Get-Service = mask Reconfigure and then
 | give the contents of %service-agent
 | then
 | send [to the agent][Request]message and then
 | receive[from the agent][Response]message and then
 | unmask Reconfigure

HBC = unfolding
 | Send-heart-beat and then
 | Start-Timed-Check Timeout
 | trap
 | | check (the datum is Okay) and then unfold
 | or
 | | check (the datum is Dead) and then
 | | Change-agent and then unfold .

Send-heart-beat = give the contents of %service-agent then
 send [to the agent][Poll]message .

Start-Timed-Check D =
 give the sum (current-time, D) then
 | patiently
 | | check (the current-time is less than it) and then
 | | Is-Ack-Present
 | or
 | | check (the current-time is not less than it) and then
 | | give Dead then escape .

Is-Ack-Present = | give the contents of %service-agent then
 | | choose [from the agent][Ack] message then remove it
 | and then
 | give Okay then escape

Change-agent = | give the contents of %service-agent then
 | give Next it
 | then
 | | send [to MP][interrupt][Reconfigure][Handler the agent]message
 | and
 | | store the agent in %service-agent .

Next Ag = □

Handler Ag = Message(Body Ag, “and then”)

Body Ag = abstraction
 | store Ag in %service-agent and then unmask Reconfigure

```

Service-agent = | unfolding
                | | receive [poll]message then
                | | | send[to sender of it][Ack]message and then unfold
                and
                | unfolding
                | | receive [Request]message then
                | | | send [to the sender of it][Response]message and then unfold .

```

5.2 Asynchronous And

The need for asynchronous transfer of control in Ada especially for mode changes has been discussed in [RTA88]. One of the proposals [Taf89] augments the select statement with an “and” clause. An example is

```

select
    delay D; Sd;
or
    accept E1; S1;
or
    accept E2; S2;
and
    S3;
end select

```

The informal meaning of this construct is as follows. On reaching the select alternative, if there is no pending entry call for the accepts or the delay is non zero, execution of statement S3 is started. However, if any of the other alternatives become open (i.e., the delay expires or an entry call is issued) before the execution of S3 is completed, the execution of the remainder of S3 is abandoned and the statement associated with the open alternative (delay/entry) is executed. In this section, we present a formal semantics for the above construct which also handles the situation where the calling task and the called task are distributed.

Since the semantics requires abandoning the current execution, when an entry call is detected, it is natural to translate an entry call as an interrupt. However, it should not affect the other tasks on the agent. Therefore, an entry call is a software interrupt. The entry call also sends the appropriate statement to be executed and other code to finish the execution of the select. The ‘continuation’ used is **trap** so that after the handler executes, the remainder of the code associated with the select is skipped. As the entries are interrupts, all statements except the select statement are impervious to interrupts. The select statement executes the “and” alternative such that it is sensitive to possible entry calls and timer interrupt. The issuer of the entry call or the timer interrupt sends a signal message to the agent executing the select statement.

The delay is modeled by a timer agent. It receives the duration of time to delay and the body to be executed when the delay expires. The timer agent is connected to the metronome in the system. It polls for the duration to exceed the specified delay and when the specified duration has elapsed it interrupts the agent that issued the delay. In keeping with the semantics of the asynchronous and, the continuation is **trap**. However, if an entry call is made before the delay expires, the timer should be reset. This is modeled by a signal **reset**.

Towards a formal description of the “and” construct we use the following abstract syntax fragment. It is not a complete semantics for the tasking model in Ada and should be considered only as an illustration. The semantic function **Establish** creates the necessary bindings in which the execution occurs. The bindings produced for the select statement

contains the set of entries and a token representing the delay statement in it (which have to be unmasked %possible-entries) and a mapping of the entry names and the delay alternative to the statement to be executed as part of the interrupt handler. The semantic function **Execute** defines the dynamic behavior of the construct. The execution of the select statement proceeds as follows. The set of entry names is obtained (via **Establish**). The timer is set to the appropriate delay and the body of the and branch (S) is started in parallel with the delay. If an entry call (a software interrupt message) is detected, the body associated with it is executed. As the and/delay alternatives must be abandoned, the continuation is **trap** (we assume that the body does not have an abnormal termination).

comment: Partial Grammar

Statement = Select | Entry-Call | Delay-Statement | □

Select = [[“select” Delay-Statement “or” Accepts “and” Statement]]

Entry-Call = [Identifier “.” Identifier]

Delay-Statement = [[“delay” Expression Statement]]

Accepts = [[“accept” Entry “;” Statement]] | [Accepts “or” Accepts]

Establish [[“select” Ds “or” As “and” S2]] =
 | Establish Ds before Establish As
 hence
 | rebind and bind %possible-entries to domain of current bindings .

Establish [[“delay” E S]] = give closure abstraction Execute S then
 bind %delay to it

Establish [[“accept” E “;” S]] = give closure abstraction
 | Reset-Timer and then Execute S
 then
 | bind (token of E) to it

Establish [[A1 “or” A2]] = Establish A1 before Establish A2

Execute [[“select” Ds “or” As “and” S2]] =
 | Establish [[“select” Ds “or” As “and” S2]]
 before
 | give the set bound to %possible-entries then
 | listening to it
 | | Execute Ds and then
 | | Execute S2 and then Reset-Timer .

Execute [[“delay” E S]] = | | Evaluate E
 and
 | | give the closure abstraction Execute S
 then
 | give Timer-Message(the datum #1,the datum #2) then
 send [to timer-agent][containing the datum] message .

Execute [[T “.” E]] = send [to agent of T][signal][token of E][For token of E] message

Body E = closure abstraction
 enact the datum bound to (token of E)

For E = Message (Body E, “trap”)

Timer-agent =

```
receive [Timer-message] message then
  give Delay(it) and
  give Body(it) and
  give sender(it)
then
  listening to set(reset)
  unfolding
    check (current-time is greater than the datum#1) and then
    send [to the datum#3][signal][%delay][containing For-delay] message
  or
    check (current-time is not greater than the datum#1) and then
    unfold
```

For-delay = Message (the datum#3, “trap”)

Reset-Timer = send [to timer-agent][signal][reset][containing Finish] message

Finish = Message(abstraction complete, “trap”)

6 Conclusion

We have shown how the effect of asynchronous transfer of control can be specified. While the transfer of control occurs at one agent, a remote agent can cause it via message passing. We have developed our ideas within the Action Notation framework. While the Action notation has been used to describe semantics for realistic programming languages, it does not support interrupts. But with a few notational additions and a change to the operational semantics, we have been able to model interrupts.

Further research is necessary to develop a high level language in which fault-tolerance can be specified. Such a language could involve constructs such as “Normal-Processing *on-fault F* Recovery” (a generalization of the asynchronous and). The work described here provides a framework in which the semantics of such constructs can be defined. The semantics of the construct can be defined by translating *F* to an interrupt and defining a handler to transfer control from “Normal-Processing” to “Recovery”.

In [Kri91], we have shown how the notation can be used to specify real-time systems. Thus the extended system can be used to describe fault-tolerant real-time systems. While we have shown two examples here, further experience is necessary to gauge the applicability of the ideas in describing the semantics of general fault-tolerant languages.

Acknowledgements

The authors thank Jens Palsberg for useful comments on the preliminary version of this paper. The authors also thank the anonymous referees for useful comments.

References

- [Avi85] A. Avizienis. The N-version Approach to Fault-Tolerant Software. *IEEE Transactions on Software Engineering*, 11(12):1491–1501, Dec 1985.
- [BN81] A.D. Birrell and B.J. Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, 2(4):39–59, February 1981.

- [CAS86] F. Cristian, H. Aghili, and R. Strong. Clock synchronization in the presence of omission and performance faults and processor joins. In *Proceedings of the Sixteenth International Symposium on Fault-Tolerant Computing*, Vienna, Austria, 1986. IEEE.
- [Cri91] F. Cristian. Understanding Fault-Tolerant Distributed Systems. *Communications of the ACM*, 34(2):56–78, February 1991.
- [HAH89] M. Hecht, J. Argon, and S. Hochhauser. A distributed fault-tolerant architecture for nuclear reactor control and safety functions. In *The 10th IEEE Real-Time Systems Symposium*, pages 214,221, 1989.
- [KK88] R. Koymans and R. Kuiper. Paradigms for real-time systems. In M. Joseph, editor, *Proceedings of the symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems: LNCS 331*, pages 159–174, Warwick, 1988. Springer Verlag.
- [Kri91] P. Krishnan. Real-time Action. In *Euromicro Workshop on Real-Time Systems*, pages 174–182, Paris, France, June 1991. IEEE.
- [KU87] J. C. Knight and J. I. A. Urquhart. On the implementation and use of Ada in a fault-tolerant distributed systems. *IEEE Trans. on Software Engineering*, 13(5):553–563, May 1987.
- [Mos82] P. D. Mosses. Abstract semantics algebras. In D. Bjoerner, editor, *Proceeding of the IFIP TC2 Working Conference on Formal Description of Programming Concepts II*, pages 63–88. North Holland, 1982.
- [Mos89] P. D. Mosses. Unified Algebras and Action Semantics. In *STACS 89, LNCS-349*, Paderborn, 1989. Springer Verlag.
- [Mos90] P. D. Mosses. Action semantics. Technical report, DAIMI: Aarhus University, 1990.
- [Mos92] P. D. Mosses. *Action Semantics*. Cambridge University Press (in the series Tracts in Theoretical Computer Science), August 1992.
- [MW87] P. D. Mosses and D. A. Watt. The use of Action Semantics. In *Proceeding of the IFIP TC2 Working Conference on Formal Description of Programming Concepts III, 1986*. North Holland, 1987.
- [RTA88] *Proceedings of the 2nd International Workshop on Real-Time Ada issues*, volume 8(7), Devon, UK, 1988. ACM, Ada Letters.
- [SG90] J. Stamos and D. Gifford. Remote evaluation. *ACM Transactions on Programming Language and Systems*, 12(4):537–565, October 1990.
- [ST87] T. K. Srikanth and S. Toueg. Optimal clock synchronization. *Journal of the Association of the Computing Machinery*, 34(3):626–645, July 1987.
- [Taf89] T. Taft. Asynchronous event handling: Revision request. Technical report, Intermetrics Inc, Boston, March 1989.
- [Wat87] D. A. Watt. An Action Semantics of standard ML. In *Mathematical Foundations of Programming Language Semantics: LNCS 298*, pages 572–598. Springer Verlag, 1987.